¿Cómo comparar objetos en Python?

Yan Orestes
12/11/2020

En este artículo te mostraré cómo funciona el operador de igualdad en **Python** y cómo cambiamos su comportamiento, controlando más nuestro código.

Tengo un sistema Python que almacena las películas que tengo en una **lista**, por una cuestión de organización, creé una función que captura todas las películas y devuelve una lista con todas ellas:

```
class Pelicula():
    def __init__(self, titulo, director):
        self.titulo = titulo
        self.director = director
    def __str__(self):
        return self.titulo + ' - ' + self.director

def captura_todas_las_peliculas():
    ## implementación de la función

mis_peliculas = captura_todas_las_peliculas()
for pelicula in mis_peliculas:
    print(pelicula)
```

Veamos cómo se ve:

```
La Teoría del Todo - James Marsh
La La Land - Damien Chazelle
El padrino - Francis Ford Coppola
```

¡Ok, tenemos nuestra lista de películas!

Después de un tiempo, quería saber si una película ya hacia parte de mi dataset. ¿Cómo podemos hacer eso?

Comparando objetos con el operador ==

Para saber si ya tengo una película, creé una función que verifica si la película que estoy buscando ya está en nuestra lista de películas:

```
def tengo_la_pelicula(pelicula_buscada):
    mis_peliculas = captura_todas_las_peliculas()
    for pelicula in mis_peliculas:
        if pelicula_buscada == pelicula:
            return True
    return False

pelicula_buscada = Pelicula('La Teoria del Todo', 'James Marsh')

if tengo_la_pelicula(pelicula_buscada):
    print('¡Tengo la pelicula!')
else:
    print('No tengo :(')
```

Ejecutamos nuestro código y este fue el resultado:

```
No tengo :(
```

¡Eh! Hemos visto que "La teoría del Todo" está en nuestra lista, entonces, ¿qué está fallando? Sabemos que el operador == funciona bien para **números y strings**:

```
x = 700
y = 700
```

```
print(x == y)

a = 'Pikachu'
b = 'Pikachu'
print(a == b)
```

El resultado:

```
True
True
```

Pero entonces, ¿por qué no funcionó como esperávamos con nuestra clase `Película`?

¿Cómo Python compara los objetos?

El punto es que cuando usamos el operador "==", Python no sabe cómo comparar objetos de tipo `Película`. ¿Qué valor se debe comparar? ¿El título? ¿El director? ¿Y de qué manera? ¡Python no lo sabe!

Con esta duda, Python elige no verificar ningún valor, sino la **identidad** de los objetos. Es decir, comprueba si las variables de comparación apuntan al mismo objeto en la memoria. Podemos verificar la identidad de un objeto usando la función **id()**:

```
a = Pelicula('La Teoria del Todo', 'James Marsh')
b = Pelicula('La Teoria del Todo', 'James Marsh')

print(id(a))
print(id(b))
```

El resultado se verá así:

```
139789394500856
139789394500800
```

Observa que, de hecho, ¡los dos valores de identidad son diferentes! Si indicamos que la variable **b** apunta al objeto de la variable **a**, el resultado sería diferente:

```
a = Película('La Teoria del Todo', 'James Marsh')
b = a

print(a == b)
```

Esta vez:

```
True
```

Esto se debe a que, en este caso, las dos variables apuntan al mismo objeto, es decir, tienen la misma identidad:

```
print(id(a))
print(id(b))
```

El resultado:

```
139824816890912
139824816890912
```

Veamos cómo funciona con los números enteros y las strings que probamos antes:

```
x = 700
y = 700

print('Los enteros:')
print(id(x))
print(id(y))

a = 'Pikachu'
b = 'Pikachu'

print('Las strings:')
print(id(a))
print(id(b))
```

Y el resultado:

```
Los enteros:
139789394500800
139789394500856
Las strings:
139789394500970
139789394500970
```

Bueno, los números tienen identidades diferentes, mientras que las strings tienen identidades idénticas. ¿Esto significa que la comparación == con strings compara el `id` y no números enteros? No exactamente...

¿Cómo Python compara los tipos primitivos?

De hecho, los id eran los mismos que las strings, pero aun así, eso no era lo que se estaba comparando. **Los tipos primitivos en Python ya han implementado de forma nativa sus propias formas de comparar**. En el caso del int y de la string, lo que se comparan son sus valores.

¿Cómo podemos cambiar este comportamiento estándar del ==, entonces? Después de todo, ¡comparar el id de los objetos no es lo que queremos! Necesitamos de alguna manera, basar nuestra comparación en algún atributo de la `Película`.

Los que vienen de otros lenguajes de programación, como Java, ya deben conocer métodos como el equals(), que se puede sobrescribir para cambiar el comportamiento. ¿Cómo funciona esto en Python?

Conociendo el rich comparison

En Python podemos implementar algo similar al `equals()`, pero aún más poderosa: la comparación rica, o como se conoce técnicamente, **rich comparison**. Con ella, podemos definir los siguientes métodos de comparación en una clase:

`__eq__()`, llamado por el operador == `__ne__()`, llamado por el operador != `__gt__()`, llamado por el operador > `__lt__()`, llamado por el operador < `__ge__()`, llamado por el operador >= `__le__()`, llamado por el operador <=

En nuestro caso, ya que estamos tratando **comparaciones de igualdad**, nos enfocaremos en el método `__eq__()`, pero es importante tener en cuenta la posibilidad de implementar

todos los tipos básicos de comparación!.

Primero necesitamos saber con qué queremos comparar. Como necesitamos que la comparación se enfoque en algo único en cada película, usaremos el título en sí. Así que implementemos:

```
class Pelicula():
    ## código omitido
    def __eq__(self, otra_pelicula):
        return self.titulo == otra_pelicula.titulo
```

Ahora que hemos aplicado esto a nuestro código, intentemos buscar una película en nuestra colección nuevamente:

```
def tengo_la_pelicula(pelicula_buscada):
    mis_peliculas = captura_todas_las_peliculas()
    for peliculas in mis_peliculas:
        if pelicula_buscada == buscada:
            return True
    return False

pelicula_buscada = Pelicula('La Teoría del Todo', 'James Marsh')

if tengo_la_pelicula(pelicula_buscada):
    print('¡Tengo la pelicula!')
else:
    print('No la tengo :(')
```

Esta vez:

```
¡Tengo la película!
```

¡Cierto!

Simplificando nuestra verificación de operador `in`

También podemos simplificar el código en nuestra función `tengo_la_pelicula()` usando el operador **in** para comprobar si la película ya está en la lista, ya que este operador también

se basa en el retorno de ==:

```
def tengo_la_pelicula(pelicula_buscada):
    mis_peliculas = captura_todas_las_peliculas()
    return pelicula_buscada in mis_peliculas
```

Peculiaridades del rich comparison

Hay algunos aspectos interesantes de nuestro código que valen la pena destacar. Primero, ten en cuenta que no hemos implementado el método **ne()** sobre el operador !=. Pero, ¿qué pasa entonces si intentamos usar este operador?

Desde Python 3, el operador `!=` devuelve automáticamente el inverso del retorno del método `__eq__()`, si el método `__ne__()` no se ha implementado.

A pesar de eso, **si tu clase hereda de otra donde el método `__ne__()` está definido (como en los tipos primitivos), el comportamiento del `!=` se basará en lo que se especifica en ese método, ¡no en el inverso de `__eq__()`!**

Por esa razón, **se recomienda declarar siempre el método `__ne__()` cuando queremos implementar el `__eq__()`**, tanto por compatibilidad con Python 2 (que sin este método comparará los ids), como para evitar posibles problemas en nuestro código.

Otro aspecto interesante de nuestro código es que **elegimos** retornar un valor boolean. ¡Sí, elegimos!

Rich comparison te permite un retorno de cualquier tipo. Esto significa que, una comparación del tipo `a > b` no necesariamente tiene que devolvernos un boolean, ya que **puede devolver cualquier cosa que implemente el desarrollador.**

En general, elegimos devolver un boolean, pero hay proyectos grandes e importantes que alteran este comportamiento, como el **SQLAlchemy**, que utiliza esta feature para facilitar la creación de queries SQL, e incluso la biblioteca **numpy**, inspiración para crear una rich comparison.

Conclusión

En esta publicación, aprendimos cómo implementar métodos de comparación para nuestras clases en Python utilizando la técnica Rich Comparison. Pudimos abordar cómo

funcionan los operadores ==, !=, >, <, >=, <=.

¡Ahora no tendremos problemas con las comparaciones entre objetos que nosotros mismos implementamos en Python!

Para continuar y aprender todo sobre Python y Ciencia de Datos, ¡mira los cursos que tenemos en **Alura**!

Para seguir expandiendo tus conocimientos descarga nuestro ebook gratuito de Data Science a través del siguiente link: https://lp.caelum.com.br/alura-latam-leads-ebook-data-science

Puedes leer también:

- Análisis de datos: analizando mi distribución con tres alternativas de visualización
- Matplotlib una biblioteca Python para generar gráficos interesantes
- Trabajando con archivos y directorios en Python

ARTÍCULOS DE TECNOLOGÍA > DATA SCIENCE

- Videos y actividades 100% en Español
  Certificado de participación
- Estudia las 24 horas, los 7 días de la semana
- Foro y comunidad exclusiva para resolver tus dudas
- Acceso a todo el contenido de la plataforma por 6 meses

¡QUIERO EMPEZAR A ESTUDIAR!

Paga en moneda local en los siguientes países

ANUAL

US$79,90

un solo pago de US$79,90

- 218 cursos
- Videos y actividades 100% en Español
- Certificado de participación

Nuevos cursos
cada semana

NAVEGACIÓN

PLANES
INSTRUCTORES
BLOG
POLÍTICA DE PRIVACIDAD
TÉRMINOS DE USO
SOBRE NOSOTROS
PREGUNTAS FRECUENTES

¡CONTÁCTANOS!

¡QUIERO ENTRAR EN CONTACTO!

BLOG

PROGRAMACIÓN
FRONT END
DATA SCIENCE
INNOVACIÓN Y GESTIÓN
DEVOPS

AOVS Sistemas de Informática S.A
CNPJ 05.555.382/0001-33

SÍGUENOS EN NUESTRAS REDES SOCIALES

ALIADOS

Empresa participante do SCALE UP ENDEAVOR